BTech CSE\IT

Code- ES 101

Programming in 'C'

Unit 1 (8 hours)

1. Introduction to Programming

a. Computer system
b. components of a computer system
c. computing environments
d. computer languages
e. creating and running programs
f. Preprocessor
g. Compilation process
h. role of linker
i. idea of invocation and execution of a programme

2. Algorithms:

a. Representation using flowcharts
b. pseudocode.

3. Introduction to C language

a. History of C
b. basic structure of C programs process of compiling and running a C program
c. C tokens
d. Keywords
e. Identifiers
f. Constants
g. Strings
h. special symbols
i. variables
j. data types
k. I/O statements
l. Interconversion of variables

4. Operators and expressions:

a. Operators
b. Arithmetic
c. relational and logical
d. assignment operators
e. increment and decrement operators
f. bitwise and conditional operators
g. special operators

h. operator precedence and associativity
i. evaluation of expressions
j. type conversions in expressions

Unit 2 (8 Hours)

1. Control structures

a. Decision statements; if and switch statement
b. Loop control statements: while, for and do while loops
c. jump statements, break, continue, goto statements

2. Arrays:

a. Concepts,
b. One dimensional array,
c. declaration and initialization of one dimensional arrays,
d. two dimensional arrays,
e. initialization and accessing
f. multi-dimensional arrays
3. Functions:
a. User defined and built-in Functions
b. storage classes,
c. Parameter passing in functions,
d. call by value

4. Passing arrays to functions:

a. idea of call by reference

5. Recursion. Strings

a. Arrays of characters
b. variable length character strings
c. inputting character strings
d. character library functions
e. string handling functions

UNIT 1

➢ Introduction to Programming

Programming is the process of creating a set of instructions (called a *program*) that tells a computer how to perform a task.

Why Learn Programming?

- Problem-solving: Break down complex tasks.
- Automation: Automate repetitive jobs.
- Creativity: Build games, apps, websites, and more.
- Career opportunities: High demand in tech and non-tech fields.

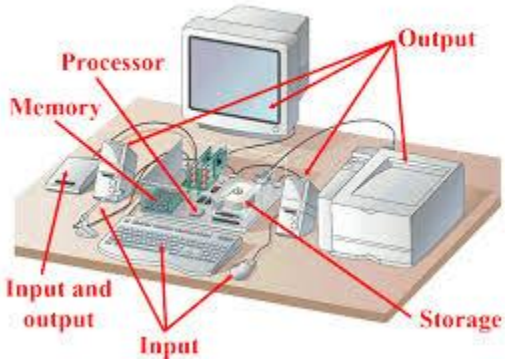
➢ Computer system

A computer system is a complete, functional setup that includes both hardware and software, working together to perform computing tasks.

A computer system consists of hardware components that have been carefully chosen so that they work well together and software components or programs that run in the computer.

The main software component is itself an operating system (OS) that manages and provides services to other programs that can be run in the computer.

In its most basic form, a computer system is a programmable electronic device that can accept input; store data; and retrieve, process and output information.



➢ Components of a computer system

A computer system is made up of hardware and software components that work together. The main hardware components include the CPU, memory (RAM and storage), motherboard, input/output devices, and the power supply. Software encompasses the operating system and applications that run on the hardware.

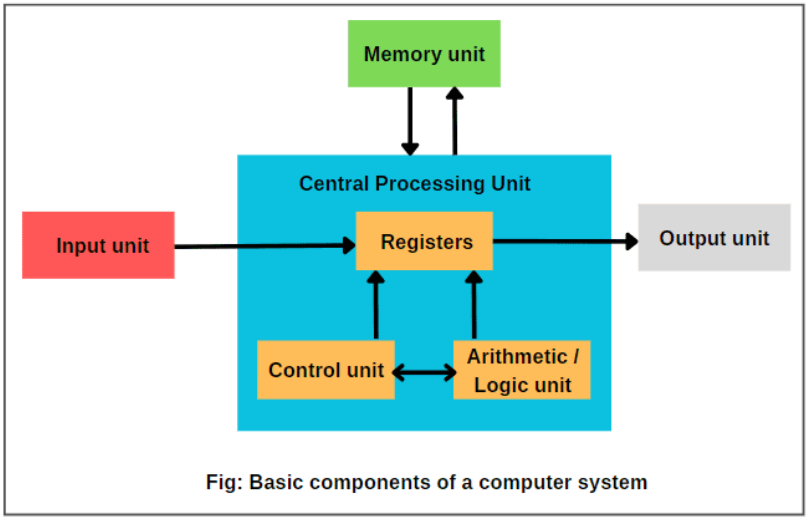


Fig: Basic components of a computer system

➢ Hardware:

- Central Processing Unit (CPU): The "brain" of the computer, responsible for executing instructions and performing calculations.
- Motherboard: The main circuit board that connects all the other components.
- Memory:
    - RAM (Random Access Memory): Temporary storage for data the CPU is actively using.
    - Storage (HDD/SSD): Permanent storage for programs and files.
- Input Devices: Allow users to enter data into the computer (e.g., keyboard, mouse, microphone).
- Output Devices: Display or present processed information (e.g., monitor, printer, speakers).
- Power Supply: Provides the necessary electrical power to the computer.
- Graphics Processing Unit (GPU): Handles graphical calculations and display.
- Other Hardware: Cooling fans, network interface cards (NIC), etc.

➢ Software:

- **Operating System (OS):**

Manages the computer's resources and provides a platform for applications (e.g., Windows, macOS, Linux).

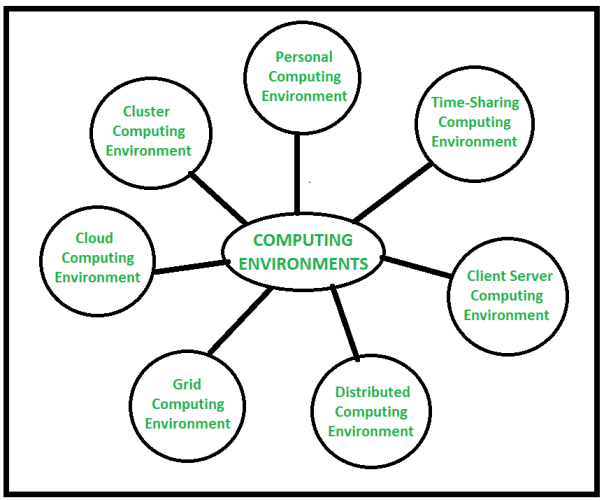
- **Application Software:**

Programs that perform specific tasks for users (e.g., word processors, web browsers, games).

➢ Computing environments

Computing environments refer to the technology infrastructure and software platforms that are used to develop, test, deploy, and run software applications. There are several types of computing environments, including:

1. Mainframe: A large and powerful computer system used for critical applications and large-scale data processing.
2. Client-Server: A computing environment in which client devices access resources and services from a central server.
3. Cloud Computing: A computing environment in which resources and services are provided over the Internet and accessed through a web browser or client software.
4. Mobile Computing: A computing environment in which users access information and applications using handheld devices such as smartphones and tablets.
5. Grid Computing: A computing environment in which resources and services are shared across multiple computers to perform large-scale computations.
6. Embedded Systems: A computing environment in which software is integrated into devices and products, often with limited processing power and memory.



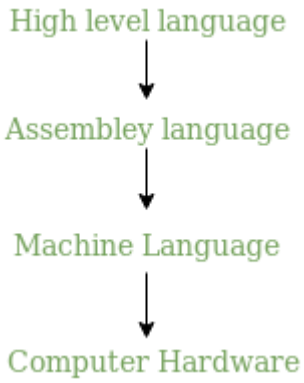
➢ Computer languages

A programming language is a set of instructions and syntax used to create software programs. Some of the key features of programming languages include:

1. **Syntax**: The specific rules and structure used to write code in a programming language.
2. **Data Types**: The type of values that can be stored in a program, such as numbers, strings, and booleans.

3. **Variables**: Named memory locations that can store values.
4. **Operators**: Symbols used to perform operations on values, such as addition, subtraction, and comparison.
5. **Control Structures**: Statements used to control the flow of a program, such as if-else statements, loops, and function calls.
6. **Libraries** and Frameworks: Collections of pre-written code that can be used to perform common tasks and speed up development.
7. **Paradigms**: The programming style or philosophy used in the language, such as procedural, object-oriented, or functional.

Hierarchy of Computer language -



Most Popular Programming Languages -

- C
- Python
- C++
- Java
- SCALA
- C#
- R
- Ruby
- Go
- Swift
- JavaScript

Advantages of programming languages:

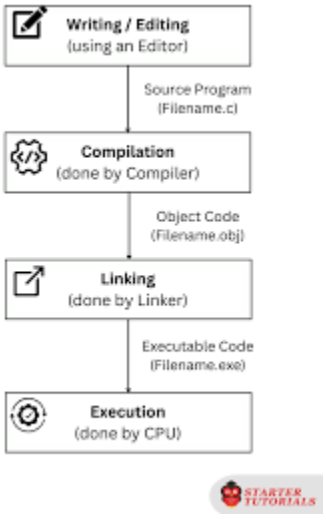
1. **Increased Productivity:** Programming languages provide a set of abstractions that allow developers to write code more quickly and efficiently.
2. **Portability:** Programs written in a high-level programming language can run on many different operating systems and platforms.
3. **Readability**: Well-designed programming languages can make code more readable and easier to understand for both the original author and other developers.
4. **Large Community:** Many programming languages have large communities of users and developers, which can provide support, libraries, and tools.

Disadvantages of programming languages:

1. **Complexity**: Some programming languages can be complex and difficult to learn, especially for beginners.
2. **Performance**: Programs written in high-level programming languages can run slower than programs written in lower-level languages.
3. **Limited Functionality**: Some programming languages may not have built-in support for certain types of tasks or may require additional libraries to perform certain functions.
4. **Fragmentation:** There are many different programming languages, which can lead to fragmentation and make it difficult to share code and collaborate with other developers.

a. Creating and Running programs

Creating and running a program involves several key steps: writing the source code, compiling it into machine code, linking it with necessary libraries, and finally executing the program. Essentially, you take human-readable instructions (the source code), convert them into a language the computer understands (machine code), and then run those instructions



1. Writing and Editing the Program:

The first step is to write the program's instructions using a text editor in a specific programming language (like C, Java, Python, etc.). The file is then saved with an appropriate extension (e.g., .c for C, .java for Java).

2. Compiling the Program:

The source code, which is in a high-level language, needs to be translated into machine code (low-level language) that the computer can directly execute. This translation is done by a compiler (a program). The compiler also checks for syntax errors in the code. If the compilation is successful, an object file (or intermediate code) is created.

3. Linking:

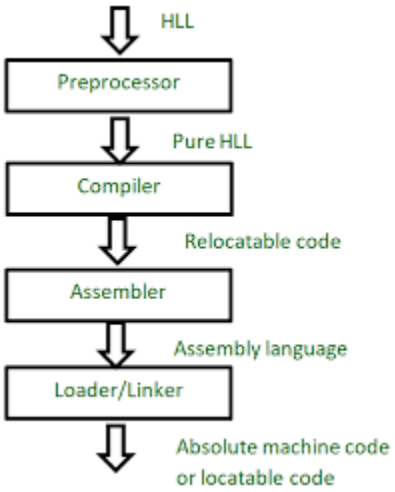
The object file may need to be combined with pre-compiled code from libraries (collections of reusable code). The linker combines these various code components to produce a single executable file.

4. Executing the Program:

Finally, the executable file is loaded into the computer's memory and the CPU starts executing the instructions. The results of the program's execution are typically displayed on the screen.

➢ Preprocessor

A preprocessor is a program that processes its input (often source code) before it's passed to another program, typically a compiler. It's a separate program invoked by the compiler during the first phase of translation. Preprocessor directives, usually starting with #, guide the preprocessor's actions, such as macro expansion, file inclusion, and conditional compilation.



➢ Key functions of a preprocessor:

- Macro Definition and Expansion:
  Preprocessor directives like #define allow you to define macros (symbolic names) that can be replaced with other text or code during preprocessing.

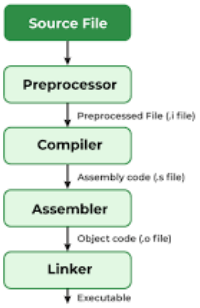
- File Inclusion:
  The #include directive inserts the contents of one file into another, often used for header files containing declarations.
- Conditional Compilation:
  Directives like #ifdef, #ifndef, #if, #else, and #endif enable the inclusion or exclusion of code sections based on conditions.
- Other Directives:
  Preprocessors also handle directives for controlling error messages, setting pragmas (compiler-specific rules), and more.

How it works:

1. The preprocessor reads the source code file.
2. It identifies preprocessor directives (lines starting with #).
3. It executes the directives, modifying the code. For example, it replaces macro names with their definitions.
4. The preprocessor generates a modified version of the source code (often with a .i extension for C/C++) that is then passed to the compiler.
5. The compiler processes this preprocessed code as if it were the original source code.

➢ Compilation process

The compilation process transforms human-readable source code into machine-executable code. It involves several stages: preprocessing, compiling, assembling, and linking. The compiler analyzes the code, checks for errors, and translates it into assembly language. The assembler then converts this assembly code into machine code, and finally, the linker combines these object files with libraries to create the final executable.

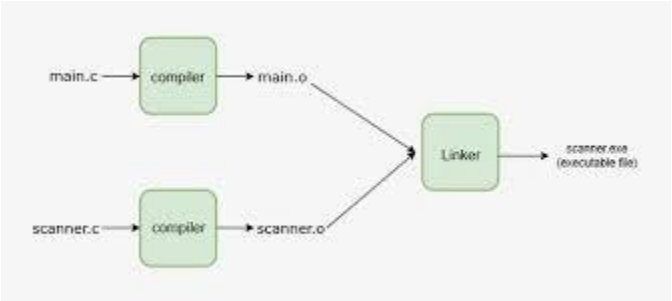


1. Preprocessing:

- The preprocessor handles directives like #include (for including header files) and #define (for macro definitions).
- It expands macros, includes header files, and performs conditional compilation.
- Essentially, it prepares the code for the actual compilation phase.
- Example: If you have #include <stdio.h>, the preprocessor replaces this line with the contents of the stdio.h header file.

➢ Role of linker

A linker is a program that combines multiple object files and libraries into a single executable file that a computer can run. It resolves references between different parts of a program, ensuring that all the necessary code and data are properly connected. Essentially, the linker bridges the gap between separate pieces of compiled code, creating a cohesive whole ready for execution.



- Input:

Linkers take object files (output from a compiler or assembler) and libraries as input.

- Combining:

The linker combines these various files into a single executable file.

- Symbol Resolution:

It resolves references between different parts of the code. For example, if one part of the code calls a function defined in another part, the linker ensures that the call is correctly directed to the actual function.

- Memory Allocation:

The linker also determines where different parts of the program will be located in memory when the program runs.

- Output:

The output of the linker is an executable file (like an .exe on Windows) that can be loaded and run by the operating system.

- Relationship to other tools:

The linker is often part of a larger toolchain that includes a compiler and/or assembler.

In essence, the linker's job is to take the different pieces of a program, put them together, and make sure they work as a single unit.

b. idea of invocation and execution of a programme

In essence, program invocation is the process of starting a program's execution, while execution is the actual carrying out of the program's instructions. Invocation is the trigger, and execution is the sequence of actions the program takes.

Execution:

- **Loading:**

The program is loaded into the computer's memory (RAM).

- **Interpretation/Compilation:**

The program's instructions (often written in a high-level language) are converted into machine code that the computer can understand.

- **Sequence:**

Instructions are then executed one by one, according to the program's logic, potentially involving calculations, data manipulation, and interaction with the operating system.

- **Output:**

The program may produce results, either displayed on the screen, stored in files, or used to influence other programs.

- **Termination:**

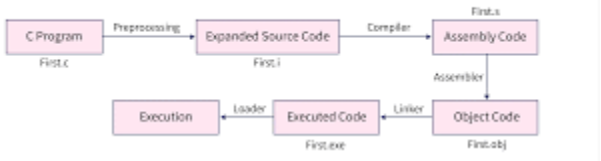
Execution concludes when the program finishes its tasks or encounters an error.

In simpler terms:

Imagine a recipe (the program). Invocation is when you decide to start following the recipe (e.g., clicking a "bake" button). Execution is the actual process of baking, following each step in the recipe (measuring ingredients, mixing, putting in the oven, etc.) until the cake is ready (the program finishes or encounters an error).

- Invocation execution model - IREE

An invocation represents a single call into a module exported function using the program state stored in a context. Users can



➢ Algorithms

An algorithm is a step-by-step set of instructions for solving a problem or completing a task. It's a finite sequence of mathematically rigorous instructions, often used in computer science and mathematics. Algorithms take input data, process it through logical rules, and produce an output, enabling computers to make decisions, process data, and automate tasks.

- Step-by-step instructions:

An algorithm is a precise and ordered sequence of instructions that a computer or other system can follow to achieve a specific outcome.

- Problem-solving:

Algorithms are designed to solve problems or perform calculations efficiently.

- Input and Output:

They typically take some form of input data, process it, and produce an output, which can be a result, a decision, or an action.

- Examples in Everyday Life:

Tying your shoes, following a recipe, and even deciding what to eat can be seen as everyday algorithms.

- Examples in Computing:

Algorithms are fundamental to computer programming, enabling tasks like searching, sorting, and more complex functions like those used in artificial intelligence.

Key Characteristics of Algorithms:

- Finiteness: An algorithm must terminate after a finite number of steps.
- Definiteness: Each step in an algorithm must be clearly defined and unambiguous.
- Effectiveness: Each step must be basic enough that it can be carried out in principle by a person using pencil and paper.
- Input: Algorithms require input data on which to operate.
- Output: Algorithms produce an output or result based on the input and the steps they follow.

Types of Algorithms:

- Sorting algorithms: Arrange data in a specific order (e.g., ascending or descending).
- Searching algorithms: Locate specific data within a larger set.
- Greedy algorithms: Make locally optimal choices at each step with the hope of finding a global optimum.
- Dynamic programming: Optimize solutions by breaking them down into smaller, overlapping subproblems.
- Backtracking algorithms: Explore different options to find a solution.
- Divide and conquer algorithms: Break down a problem into smaller subproblems, solve them recursively, and then combine the solutions.

Why are algorithms important?

- Efficiency: Algorithms enable computers to perform tasks quickly and efficiently.
- Automation: They automate repetitive tasks, freeing up human resources.
- Consistency: Algorithms produce consistent results, ensuring reliability.

- Scalability: They can be adapted to handle large amounts of data or complex problems.
- Foundation of Technology: Algorithms are the building blocks of modern technology, powering everything from search engines to artificial intelligence.

➢ Representation using flowcharts

Flowcharts are visual representations of processes or algorithms, using standardized symbols to show the sequence of steps and decisions. They are used to analyze, design, document, and manage processes or programs in various fields.

Key aspects of flowcharts:

- Visual Representation:

Flowcharts use shapes (like ovals, rectangles, diamonds) and arrows to represent different actions, decisions, and the flow of information.

- Step-by-step process:

They depict the sequence of steps in a process, making it easy to understand the flow of operations from start to finish.

- Standardized symbols:

Using standard symbols ensures clarity and consistency across different users and applications.

- Applications:

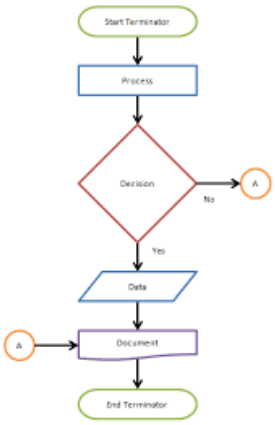
Flowcharts are used in various fields, including software development, business process modeling, and engineering.

- Benefits:

Flowcharts enhance communication, facilitate problem-solving, and aid in documentation and training

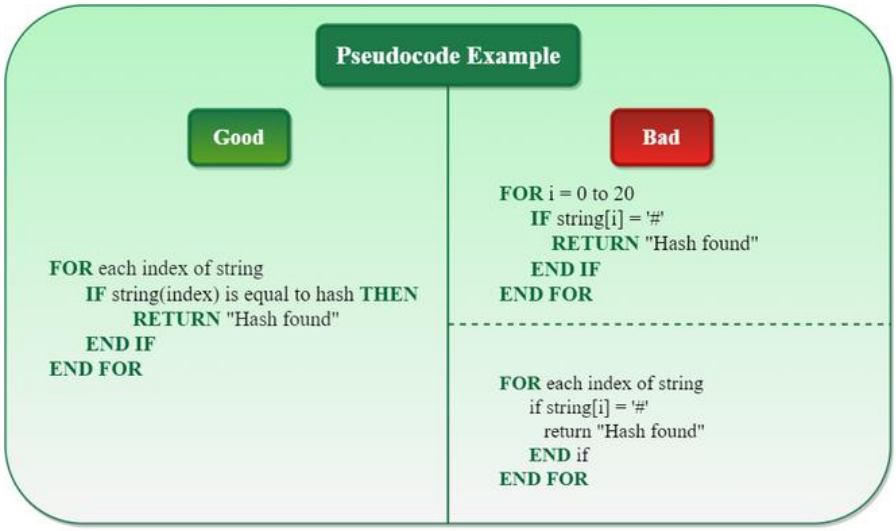
Common flowchart symbols and their meanings:

- Oval (or Terminator): Represents the start and end points of the process.
- Rectangle: Represents a process step or action.
- Diamond: Represents a decision point, where a choice needs to be made.
- Parallelogram: Represents input or output operations.
- Arrow: Indicates the direction of the flow of the process.



➢ Pseudocode

Pseudocode is a high-level description of a computer program's logic, using a mixture of natural language and programming-like syntax, but without the strict rules of any specific programming language. It's essentially a human-readable representation of an algorithm, intended to be easily understood by both programmers and non-programmers.



Key Characteristics of Pseudocode:

- **Plain Language:**

Uses simple, clear English or other natural language phrases to describe the steps of an algorithm.

- **Programming Constructs:**

Employs keywords like "IF," "THEN," "ELSE," "FOR," "WHILE," "INPUT," and "OUTPUT" to represent control flow and data manipulation.

- **No Strict Syntax:**

Doesn't adhere to the specific syntax rules of any programming language, allowing for flexibility and readability.

- **Focus on Logic:**

Concentrates on the logic of the algorithm rather than the details of its implementation in a particular language.

- **Readability:**

Designed to be easily understood by anyone involved in the development process, including designers, coders, and project managers.

- **Not Executable:**

Pseudocode cannot be directly executed by a computer; it's a planning tool, not a program.

Purpose of Pseudocode:

- **Algorithm Design:**

Helps programmers plan and organize the logic of a program before writing actual code.

- **Communication:**

Facilitates communication between programmers and stakeholders, regardless of their programming language expertise.

- **Problem Solving:**

Provides a structured way to break down complex problems into smaller, manageable steps.

- **Code Efficiency:**

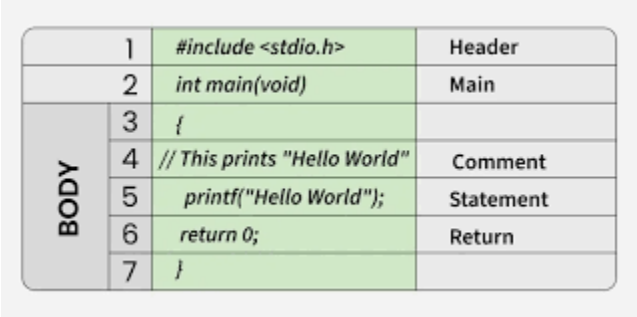
Can help identify potential errors or logical flaws in the algorithm early in the development process.

Introduction to C language

C is a general-purpose procedural programming language initially developed by Dennis Ritchie in 1972 at Bell Laboratories of AT&T Labs. It was mainly created as a system programming language to write the UNIX operating system.

Why Learn C?

- C is considered mother of all programming languages as many later languages like Java, PHP and JavaScript have borrowed syntax/features directly or indirectly from the C.
- If a person learns C programming first, it helps to learn any modern programming language as it provide a deeper understanding of the fundamentals of programming and underlying architecture of the operating system like pointers, working with memory locations etc.
- C is widely used in operating systems, embedded systems, compilers, databases, networking, game engines, and real-time systems for its efficiency to work in low resource environment and hardware-level support.



History of C

C, a general-purpose programming language, was developed in the early 1970s by Dennis Ritchie at Bell Labs. It was designed as an evolution of the B and BCPL languages and became integral to the development of the Unix operating system. C's efficiency, portability, and ability to provide low-level control have made it a foundational language in computer science, influencing many subsequent programming languages.

Here's a more detailed look at the history of C:

- Early Development:

C was created at Bell Labs in the early 1970s by Dennis Ritchie, building upon the work of earlier languages like BCPL and B.

- Unix Integration:

C's design was closely tied to the development of the Unix operating system, and it was used to rewrite the Unix kernel from assembly language.

- Popularity:

C gained significant popularity due to its efficiency and portability, making it a versatile language for various applications.

- Standardization:

C was standardized by ANSI in 1989 and later by ISO in 1990, ensuring consistency and broader adoption.

- Continued Relevance:

Despite its age, C remains widely used, particularly in systems programming, embedded systems, and where performance and control are critical.

- Influence on Other Languages:

C's design principles have influenced many subsequent programming languages, including C++, Java, and Python.

- C - Wikipedia

C is a general-purpose programming language.

Basic structure of C programs process of compiling and running a C program

A C program typically follows a structured format consisting of several key sections, including documentation, preprocessor directives, global declarations, the main function, and potentially user-defined functions. Compiling a C program involves several stages: preprocessing, compilation, assembly, and linking, which ultimately transforms the source code into an executable file.

Basic Structure of a C Program:

1. 1. Documentation Section:

Includes comments explaining the program's purpose, author, and other relevant information.

2. 2. Preprocessor Section:

Contains preprocessor directives, starting with #, which instruct the compiler to perform actions like including header files (e.g., #include <stdio.h>) or defining constants.

3. 3. Definition Section:

Defines symbolic constants using #define directives.

4. 4. Global Declaration Section:

Declares variables that can be accessed by multiple functions within the program.

5. 5. main() Function:

The entry point of the program, where execution begins. It has a declaration part (for variables) and an execution part (containing statements).

6. 6. User-defined Functions:

Sections for functions that perform specific tasks, called by the main() function.

Compiling and Running a C Program:

1. Source Code Creation:

The program is written in a text editor and saved with a .c extension.

2. Preprocessing:

The preprocessor expands macros, includes header files, and handles conditional compilation directives.

3. Compilation:

The compiler translates the preprocessed code into assembly language or machine code, producing an object file (.o or .obj).

4. Assembly:

The assembler converts the assembly code into machine code, creating an object file.

5. Linking:

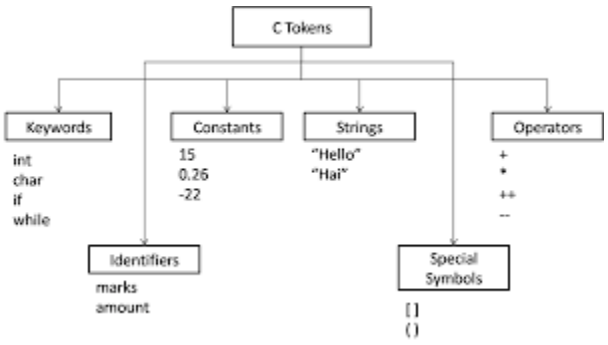
The linker combines the object file with necessary library functions and resolves external references, creating the final executable file (e.g., a.out or .exe).

6. Execution:

The operating system loads the executable file into memory and starts its execution.

➢ Tokens

In C programming, tokens are the basic building blocks of a program, representing the smallest individual units recognized by the compiler. They are the individual words, punctuation marks, and other elements that the compiler uses to understand the structure and meaning of the code.



Here's a breakdown of the different types of C tokens:

- Keywords: Reserved words with predefined meanings in C (e.g., int, if, while).
- Identifiers: Names given to variables, functions, and other entities by the programmer.
- Constants: Fixed values that do not change during program execution (e.g., integer constants like 10, floating-point constants like 3.14, character constants like 'a').
- Strings: Sequences of characters enclosed in double quotes (e.g., "hello").
- Operators: Symbols that perform operations on operands (e.g., +, -, *, /, =, ==).

- Special Symbols: Characters that have specific meanings in C, such as parentheses (), braces {}, commas ,, and semicolons ;

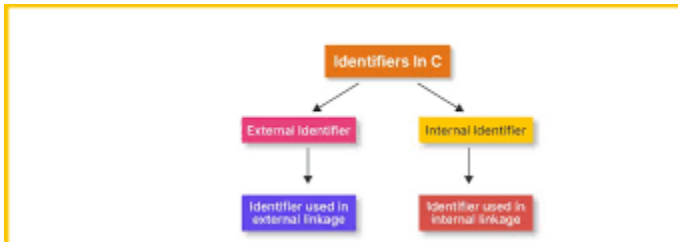
Keywords

Keywords in C are **reserved words** that have special meanings to the compiler. You **cannot use them as variable names**, function names, or identifiers. They form the foundation of C programming by defining the **syntax and structure** of the language.

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

Identifiers

In C, identifiers are user-defined names used to identify variables, functions, arrays, and other program elements. They are essentially the way you label and refer to different parts of your code. Identifiers must adhere to specific rules, including starting with a letter or underscore, being composed of alphanumeric characters and underscores, and not exceeding 31 characters in length.



Rules for Identifiers:

- First Character: An identifier must start with either a letter (A-Z or a-z) or an underscore (_), according to Unstop.
- Characters: After the first character, identifiers can contain letters, digits (0-9), and underscores.
- Length: Identifiers have a maximum length of 31 characters according to Unstop.

- Case Sensitivity: C is case-sensitive, meaning myVariable and myvariable are treated as different identifiers according to Dr. Shyama Prasad Mukherjee University.
- No Spaces or Special Characters: Identifiers cannot contain spaces or special characters like !, @, #, etc.
- No Keywords: You cannot use C keywords (like int, for, if, etc.) as identifiers, according to Learn Microsoft.

Examples:

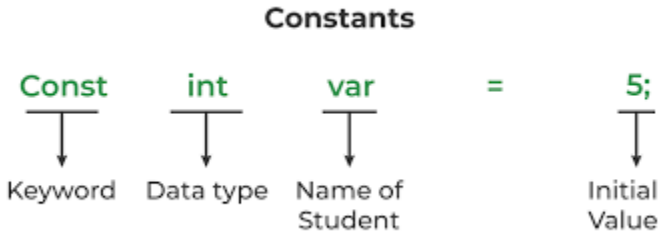
- Valid identifiers: x, count, student_name, _result, myVariable
- Invalid identifiers: 123count (starts with a digit), student name (contains a space), if (a keyword)

Importance:

- Identifiers are essential for organizing and accessing data and functions within a program.
- They allow you to refer to specific variables, functions, or other program elements in your code.
- Using meaningful and descriptive identifiers enhances code readability and maintainability.
- C tokens - keywords, identifiers, constants, operators, special ...

Constants

In C, constants are values that remain unchanged throughout the execution of a program. They are classified into several types, including integer constants, floating-point constants, character constants, string constants, and symbolic constants.



Types of Constants in C:

1. Integer Constants:

These represent whole numbers, both positive and negative, without any fractional or decimal part. Examples include 10, -5, 0, 1000.

2. Floating-Point Constants:

These represent numbers with fractional values, including a decimal point. Examples include 3.14, -0.005, 1.23e4. Floating-point constants can be specified with an optional suffix f or F to denote float type, or l or L to denote long double type. Without a suffix, they are of type double.

3. Character Constants:

These represent a single character enclosed in single quotes, like 'A', '7', or '%'. Each character has an associated integer value (ASCII code).

4. String Constants:

These represent sequences of characters enclosed in double quotes, like "Hello, world!".

5. Symbolic Constants:

These are defined using #define preprocessor directive or the const keyword. #define creates a macro substitution, while const creates a read-only variable.

Strings

In C programming, "strings" are essentially arrays of characters terminated by a null character (\0). This null character signifies the end of the string.

Here's a breakdown of key aspects:

- Declaration and Initialization:
    - Using character arrays: char str[] = "Hello"; The compiler automatically adds the null terminator.
    - Using pointers: char *str = "World"; This creates a pointer to a string literal, which is typically stored in read-only memory. Modifying such a string directly will result in a runtime error.
- Storage:
    - Stack: Strings declared as char str[] are stored on the stack and are modifiable.
    - Read-only memory (for string literals): Strings initialized with string literals (e.g., char *str = "Hello";) are stored in a read-only segment of memory.
    - Heap (dynamic allocation): Strings can be dynamically allocated on the heap using malloc and related functions, allowing for flexible size and lifetime management.
- Input and Output:
    - printf: Used to display strings using the %s format specifier.
    - scanf: Used to read strings, typically using %s. Be cautious with scanf("%s", ...) as it stops at the first whitespace and can lead to buffer overflows if the input is longer than the allocated array size.
    - fgets: A safer alternative for reading strings, allowing you to specify the maximum number of characters to read, preventing buffer overflows.
- String Manipulation Functions (from string.h):
    - strlen(): Returns the length of a string (excluding the null terminator).
    - strcpy(): Copies one string to another.
    - strcat(): Concatenates (joins) two strings.
    - strcmp(): Compares two strings lexicographically.
    - strstr(): Finds the first occurrence of a substring within a string.
    - strchr(): Finds the first occurrence of a character within a string.

➢ Special symbols

Special symbols in C are characters that hold specific meaning and functionality within the language's syntax and structure. They are distinct from alphanumeric characters and play crucial roles in defining program logic, operations, and organization.

Common examples of special symbols and their uses include:

- Punctuation:
    - Semicolon (;): Terminates statements.
    - Comma (,): Separates elements in lists (e.g., variable declarations, function arguments).
    - Parentheses (): Used for function calls, grouping expressions, and defining function parameters.
    - Curly Braces {}: Define blocks of code (e.g., function bodies, control flow statements).
    - Square Brackets []: Used for array indexing and declaring array types.
- Operators:
    - Arithmetic Operators (+, -, *, /, %): Perform mathematical calculations.
    - Assignment Operator (=): Assigns a value to a variable.
    - Relational Operators (<, >, <=, >=, ==, !=): Compare values.
    - Logical Operators (&&, ||, !): Combine or negate boolean expressions.
    - Bitwise Operators (&, |, ^, ~, <<, >>): Perform operations on individual bits.
    - Increment/Decrement Operators (++, --): Increase or decrease a variable's value by one.
    - Address-of Operator (&): Returns the memory address of a variable.
    - Dereference Operator (*): Accesses the value at a memory address (used with pointers).
- Other Special Characters:
    - Hash (#): Initiates preprocessor directives (e.g., #include, #define).
    - Dot (.): Accesses members of structures or unions.
    - Backslash (`\`): Used for escape sequences (e.g., \n for newline, \t for tab).

➢ Variables

In C, variables are named storage locations in memory that hold data values. They are fundamental for storing and manipulating information within a program. Each variable has a specific data type, which determines the kind of data it can hold (e.g., integers, floating-point numbers, characters), and a unique name that allows you to access and modify its value.

Key aspects of variables in C:

- Declaration:

Variables must be declared before use, specifying their data type and name. For example, int age; declares an integer variable named age.

- Data Types:

Common data types include int (integers), float (floating-point numbers), char (characters), and double (double-precision floating-point numbers).

- Initialization:

Variables can be initialized with a value during declaration, or later in the program. For example, int age = 30; declares and initializes age to 30.

- Naming Rules:

Variable names must start with a letter or underscore, followed by letters, numbers, or underscores. C is case-sensitive, so age and Age are treated as different variables.

- Scope:

The scope of a variable determines where it can be accessed within the program. Local variables are declared inside a function and are only accessible within that function, while global variables are declared outside functions and are accessible throughout the program.

- **Examples:**
    - int count = 10; declares an integer variable count and initializes it to 10.
    - float price; declares a floating-point variable price.
    - char grade = 'A'; declares a character variable grade and initializes it to 'A'.
    - double distance; declares a double-precision floating-point variable distance.
    - int x, y, z; declares three integer variables x, y, and z.

Variables are containers for storing data values, like numbers and characters.

In C, there are different types of variables (defined with different keywords), for example:

- int - stores integers (whole numbers), without decimals, such as 123 or -123
- float - stores floating point numbers, with decimals, such as 19.99 or -19.99
- char - stores single characters, such as 'a' or 'B'. Characters are surrounded by single quotes

Declaring (Creating) Variables

To create a variable, specify the type and assign it a value:

Syntax

type variableName = *value*;

Where *type* is one of C types (such as int), and *variableName* is the name of the variable (such as x or myName). The equal sign is used to assign a value to the variable.

So, to create a variable that should store a number, look at the following example:

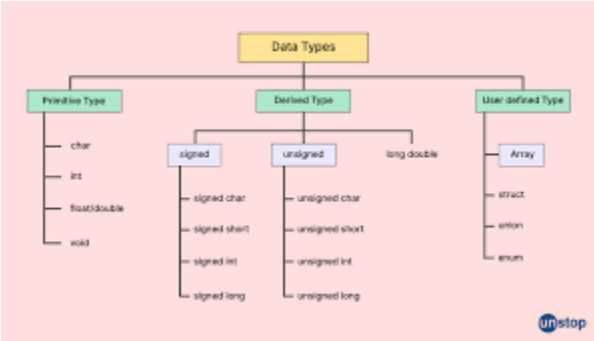
Example

Create a variable called myNum of type int and assign the value 15 to it:

int myNum = 15;

➢ Data types

In C, data types define the type of data a variable can hold, such as integers, characters, or floating-point numbers. They dictate how much memory is allocated for the variable and how the data is interpreted. C has several built-in data types: int, char, float, double, and void. Additionally, short, long, signed, and unsigned can be used to modify the basic types.



1. Basic Data Types:

- int:

Used to store integers (whole numbers). For example, int age = 30;.

- char:

Used to store single characters (letters, symbols, etc.). Characters are enclosed in single quotes (e.g., char letter = 'a';).

- float:

Used to store single-precision floating-point numbers (numbers with decimal points). For example, float pi = 3.14159;.

- double:

Used to store double-precision floating-point numbers, providing more accuracy than float. For example, double price = 19.99;.

- void:

Indicates that no value is associated with the variable. It's often used for function return types when a function doesn't return any value.

2. Type Modifiers:

- short and long:

Modify the size of integer types. short typically uses less memory than int, while long may use more memory than int depending on the system.

- signed and unsigned:

Determine whether a data type can hold negative values. signed allows both positive and negative numbers, while unsigned only allows positive numbers and zero.

3. Derived Data Types:

- Arrays: Collections of elements of the same data type.
- Pointers: Variables that store memory addresses.
- Structures: Used to group variables of different data types.
- Unions: Structures where all members share the same memory location.

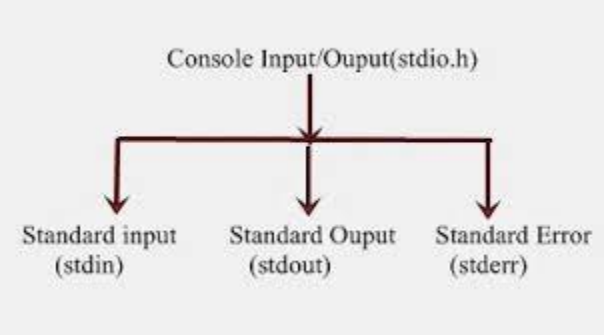
4. User-Defined Data Types:

- Enumerations (enums): Used to create named integer constants.

The choice of data type depends on the nature of the data being stored and the precision required. For example, if you need to store a large integer value, you might choose long int or long long int. If you need to store a decimal number with high accuracy, you would choose double over float.

I/O statements

C programming, Input/Output (I/O) statements are used to handle the flow of data between a program and external devices, such as the console (keyboard and screen) or files. These statements are categorized into formatted and unformatted I/O functions.



Formatted I/O Functions:

These functions allow for controlled input and output using format specifiers to define the data type and presentation.

- **printf(): Used to print formatted output to the console.**

```
#include <stdio.h>
int main() {
   int num = 10;
   printf("The number is: %d\n", num);
   return 0;
}
```

- **scanf(): Used to read formatted input from the console.**

```
#include <stdio.h>
int main() {
   int num;
   printf("Enter a number: ");
   scanf("%d", &num);
   printf("You entered: %d\n", num);
   return 0;
}
```

- fprintf(): Used to write formatted output to a file.
- fscanf(): Used to read formatted input from a file.

Unformatted I/O Functions:

These functions handle single characters or strings without specific formatting.

- getchar(): Reads a single character from the console.
- putchar(): Writes a single character to the console.
- gets(): Reads a string from the console (caution: potentially unsafe, prefer fgets()).
- puts(): Writes a string to the console, followed by a newline.
- fgetc(): Reads a single character from a file.
- fputc(): Writes a single character to a file.
- fgets(): Reads a string from a file or the console (safer than gets()).
- fputs(): Writes a string to a file.

File I/O Functions:

These functions specifically handle operations with files.

- fopen(): Opens a file.
- fclose(): Closes a file.
- fread(): Reads data from a file.

- fwrite(): Writes data to a file.

➢ Interconversion of variables

Interconversion of variables refers to the process of changing a variable from one type to another. This can involve explicit type conversion, where the programmer specifies the conversion, or implicit type conversion, where the system automatically handles the change. Examples include converting an integer to a float, or changing a string to a number.

Types of Interconversion:

- **Explicit Conversion:**

This is when the programmer directly instructs the system to convert a variable's data type. This is often done using casting operators (e.g., (int), (float)).

- **Implicit Conversion:**

In this case, the system automatically converts the data type based on the context of the operation. For instance, if an integer is assigned to a floating-point variable, the system will implicitly convert the integer to a float.

Reasons for Interconversion:

- **Preventing Data Loss:**

Converting a variable to a more precise data type (e.g., from float to double) can help prevent loss of information during calculations.

- **Compatibility:**

When performing operations on variables of different data types, it may be necessary to convert them to a compatible type for the operation to be valid.

- **Specific Requirements:**

Some functions or operations may require specific data types as input, necessitating conversion.

Examples:

- **Integer to Float:** int num = 10; float floatNum = (float) num;
- **Float to Integer:** float num = 10.5; int intNum = (int) num; (This will truncate the decimal part)
- **String to Integer:** string numStr = "123"; int num = stoi(numStr); (Requires a string conversion function, e.g., stoi in C++)

In essence, interconversion of variables is a fundamental concept in programming that allows for flexibility and proper handling of data during computations and operations.

Operators and expressions:

- Operators

In programming, operators are special symbols that perform operations on values (operands). An expression is a combination of operands and operators that evaluates to a single value. Essentially, operators dictate what actions are performed on data (operands) to produce a result, and expressions represent these actions within a program

- Operators:

Symbols or keywords that perform operations. Examples include + (addition), - (subtraction), * (multiplication), / (division), == (equality), && (logical AND), etc.

- Operands:

The values or variables on which operators act. For example, in the expression 5 + 3, 5 and 3 are operands.

- Expressions:

Combinations of operands and operators that evaluate to a single value. For example, 2 * (x + y) is an expression.

Types of Operators:

- Arithmetic Operators: Perform mathematical calculations (e.g., +, -, *, /, %).
- Logical Operators: Combine or modify boolean values (e.g., &&, ||, !).
- Relational Operators: Compare values and return a boolean result (e.g., ==, !=, >, <).
- Assignment Operators: Assign values to variables (e.g., =).
- Bitwise Operators: Perform operations on individual bits of data.
- Unary Operators: Operate on a single operand (e.g., - in -5).
- Binary Operators: Operate on two operands (e.g., + in a + b).

Example:

In the expression x = (a + b) * c, x, a, b, and c are operands. The operators are =, +, and *. This expression first adds a and b, then multiplies the result by c, and finally assigns the final value to the variable x.

Arithmetic Operators:

- Used for mathematical calculations.
- Examples: + (addition), - (subtraction), * (multiplication), / (division), % (modulo).
- Example: x + y adds the values of x and y.

2. Comparison Operators:

- Used to compare values and return a Boolean result (true or false).
- Examples: == (equal to), != (not equal to), > (greater than), < (less than), >= (greater than or equal to), <= (less than or equal to).
- Example: x > y checks if x is greater than y.

3. Logical Operators:

- Used to combine or modify Boolean expressions.
- Examples: && (AND), || (OR), ! (NOT).

- Example: (x > 5) && (y < 10) is true only if both conditions are true.

4. Bitwise Operators:

- Used to perform operations at the bit level.
- Examples: & (bitwise AND), | (bitwise OR), ^ (bitwise XOR), ~ (bitwise NOT), << (left shift), >> (right shift).

5. Assignment Operators:

- Used to assign a value to a variable.
- Examples: = (assignment), += (add and assign), -= (subtract and assign), *=, /=, %=, etc.
- Example: x += 5 is equivalent to x = x + 5.

6. Other Operators:

- Concatenation Operators: Used to join strings.
- Increment/Decrement Operators: Used to increase or decrease a variable's value by one.
- Ternary Operator: A shorthand for if-else statements.

➢ Arithmetic

Arithmetic in C language involves performing mathematical calculations using a set of predefined operators. These operators are used to manipulate numerical data types, such as integers and floating-point numbers.

Basic Arithmetic Operators:

- **Addition (+): Adds two operands.**

```
int sum = 10 + 5; // sum will be 15
```

- **Subtraction (-): Subtracts the second operand from the first.**

```
int difference = 10 - 5; // difference will be 5
```

- **Multiplication (*): Multiplies two operands.**

```
int product = 10 * 5; // product will be 50
```

- **Division (/): Divides the numerator by the denominator. For integer division, it truncates the decimal part.**

```
int quotient_int = 10 / 3; // quotient_int will be 3
float quotient_float = 10.0 / 3.0; // quotient_float will be approximately 3.33
```

- **Modulo (%): Returns the remainder after an integer division.**

```
int remainder = 10 % 3; // remainder will be 1
```

Increment and Decrement Operators:

- Increment (++): Increases the value of a variable by one.

```
int num = 5;
num++; // num becomes 6
```

- Decrement (--): Decreases the value of a variable by one.

```
int num = 5;
num--; // num becomes 4
```

➢ Relational and logical

Relational operators are used to compare two operands and determine the relationship between them. They always return a Boolean value, which is represented as 1 for true and 0 for false in C.

- **Equal to (==): Checks if two operands are equal.**

```
int a = 5, b = 5;
if (a == b) { // This condition is true
    // ...
}
```

- **Not equal to (!=): Checks if two operands are not equal.**

```
int a = 5, b = 10;
if (a != b) { // This condition is true
    // ...
}
```

- **Less than (<): Checks if the left operand is less than the right operand.**

```
int a = 5, b = 10;
if (a < b) { // This condition is true
    // ...
}
```

- **Greater than (>): Checks if the left operand is greater than the right operand.**

```
int a = 10, b = 5;
if (a > b) { // This condition is true
   // ...
}
```

- **Less than or equal to (<=): Checks if the left operand is less than or equal to the right operand.**

```
int a = 5, b = 5;
if (a <= b) { // This condition is true
   // ...
}
```

- **Greater than or equal to (>=): Checks if the left operand is greater than or equal to the right operand.**

```
int a = 10, b = 5;
if (a >= b) { // This condition is true
   // ...
}
```

➢ Logical Operators:

Logical operators are used to combine or manipulate Boolean expressions (expressions that evaluate to true or false). They also return a Boolean value (1 for true, 0 for false).

- **Logical AND (&&): Returns true if both operands are true.**

```
int x = 1, y = 0;
if (x && y) { // This condition is false
   // ...
}
```

- **Logical OR (||): Returns true if at least one operand is true.**

```
int x = 1, y = 0;
if (x || y) { // This condition is true
   // ...
}
```

- **Logical NOT (!): Reverses the logical state of its operand. If an operand is true, ! makes it false, and vice-versa.**

```
int x = 1;
if (!x) { // This condition is false
```

```
    // ...
}
```

➢ Assignment operators

Assignment operators are used to assign values to variables. The most basic assignment operator is the equals sign (=), which assigns the value on the right side to the variable on the left side. For example, x = 5 assigns the value 5 to the variable x. There are also compound assignment operators that combine arithmetic operations with assignment, like +=, -=, *=, and /=, which provide shorthand ways to modify a variable's value.

Basic Assignment (=):

The fundamental assignment operator, assigning the value on the right to the variable on the left.

- Example: int age = 30;

Addition Assignment (+=):

Adds the right operand to the left operand and assigns the result to the left operand.

- **Example: x += 5; (same as x = x + 5;)**

Subtraction Assignment (-=):

Subtracts the right operand from the left operand and assigns the result to the left operand.

- Example: y -= 2; (same as y = y - 2;)

Multiplication Assignment (=):

Multiplies the left operand by the right operand and assigns the result to the left operand.

- Example: z *= 4; (same as z = z * 4;)

Division Assignment (/=):

Divides the left operand by the right operand and assigns the result to the left operand.

- Example: a /= 2; (same as a = a / 2;)

Modulo Assignment (%=):

Calculates the modulo (remainder) of the left operand divided by the right operand and assigns the result to the left operand.

- Example: b %= 3; (same as b = b % 3;)

Bitwise Assignment Operators:

These operators combine bitwise operations (AND, OR, XOR, left shift, right shift) with assignment.

- Example: x &= y; (same as x = x & y;)

These operators offer a concise way to modify variables in programming, making code more readable and efficient.

Assignment operators are used to assign values to variables. In the example below, we use the assignment operator (=) to assign ...

increment and decrement operators

Increment and decrement operators are unary operators used to increase or decrease the value of a variable by one. The increment operator (++) adds one to its operand, while the decrement operator (--) subtracts one from it. These operators can be used in both prefix (e.g., ++x) and postfix (e.g., x++) forms, with subtle differences in their behavior within expressions

Increment Operator (++)

- Prefix Increment (++x): The variable is incremented before its value is used in the expression.
- Postfix Increment (x++): The variable's current value is used in the expression, and then the variable is incremented.

Decrement Operator (--)

- Prefix Decrement (--x): The variable is decremented before its value is used in the expression.
- Postfix Decrement (x--): The variable's current value is used in the expression, and then the variable is decremented

➢ bitwise and conditional operators

Bitwise operators manipulate individual bits within integers, while conditional operators (also known as ternary operators) provide a concise way to express if-else logic in a single line

Bitwise Operators:

- AND (&): Compares corresponding bits of two operands. If both bits are 1, the result is 1; otherwise, it's 0.
- OR (|): Compares corresponding bits. If at least one bit is 1, the result is 1; otherwise, it's 0.
- XOR (^): Compares corresponding bits. If the bits are different, the result is 1; otherwise, it's 0.
- NOT (~): Inverts each bit of an operand (0 becomes 1, and 1 becomes 0).

- Left Shift (<<): Shifts bits to the left, filling with zeros on the right.
- Right Shift (>>): Shifts bits to the right, filling with zeros (or the sign bit, depending on the implementation) on the left.

Special operators

Special operators are symbols in programming languages that perform specific, often unique, tasks beyond basic arithmetic, comparison, or assignment. They can manipulate data in various ways, including memory access, control flow, or object manipulation.

Common Special Operators:

- Comma Operator (,): In some languages (like C/C++), the comma operator allows multiple expressions to be evaluated sequentially, with the result of the last expression being the value of the entire expression. This can be used to combine multiple operations into a single statement.
- sizeof Operator: Used to determine the size (in bytes) of a variable or data type at compile time.
- Address-of Operator (&): Returns the memory address of a variable. This is crucial for pointer manipulation in languages like C/C++.
- Dereference Operator ():*: Used with pointers to access the value stored at the memory address held by a pointer.
- Conditional/Ternary Operator (?:): A shorthand way to write an if-else statement in a single line. For example, in Java or C++, result = condition ? value_if_true : value_if_false;.
- Instanceof (Java): Used to check if an object is an instance of a particular class or interface.
- Dot Operator (.): Used to access members (fields or methods) of an object or structure.
- Arrow Operator (->): Used to access members of a structure or class through a pointer.
- new Operator (Java): Used to create new objects dynamically.
- Array subscript operator (): Used to access elements within an array.

Special Operators in Different Languages:

- C/C++: sizeof, &, *, . , ->, ,.
- Java: instanceof, . , [], new.
- SQL: LIKE, IN, BETWEEN, IS NULL, EXISTS, UNION.
- Python: is, is not, in, not in (primarily for identity and membership checks).

➢ Operator precedence and associativity

Operator precedence and associativity are rules that dictate the order in which operations are performed in expressions. Precedence determines which operators are evaluated first (e.g., multiplication before addition), while associativity dictates the evaluation order for operators with the same precedence (e.g., left-to-right for addition and subtraction).

Operator Precedence:

- Precedence defines the priority of operators. Operators with higher precedence are evaluated before those with lower precedence.
- For example, in the expression 2 + 3 * 4, multiplication (*) has higher precedence than addition (+), so it's evaluated first, resulting in 2 + 12 = 14.
- Parentheses can be used to override precedence and force a specific order of evaluation.

Operator Associativity:

- When an expression contains multiple operators with the same precedence, associativity determines the order in which they are evaluated.
- Left-to-right associativity: Operators are evaluated from left to right (e.g., 10 - 5 - 2 is evaluated as (10 - 5) - 2).
- Right-to-left associativity: Operators are evaluated from right to left (e.g., a = b = c is evaluated as a = (b = c)).
- Most arithmetic, comparison, and logical operators have left-to-right associativity.
- The assignment operator (=) and the unary operators (like - in -a) are right-to-left associative.

OPERATOR	TYPE	ASSOCIATIVITY
() [] . ->		left-to-right
++ -- + - ! ~ (type) * & sizeof	Unary Operator	right-to-left
* / %	Arithmetic Operator	left-to-right
+ -	Arithmetic Operator	left-to-right
<< >>	Shift Operator	left-to-right
< <= > >=	Relational Operator	left-to-right
== !=	Relational Operator	left-to-right
&	Bitwise AND Operator	left-to-right
^	Bitwise EX-OR Operator	left-to-right
|	Bitwise OR Operator	left-to-right
&&	Logical AND Operator	left-to-right
||	Logical OR Operator	left-to-right
? :	Ternary Conditional Operator	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Operator	right-to-left
,	Comma	left-to-right

Example:

In the expression a + b * c, * has higher precedence than +, so b * c is evaluated first. If * and / have the same precedence, and they are next to each other, the associativity (left-to-right) determines that the leftmost operator is evaluated first. So, a / b * c would be evaluated as (a / b) * c

- **Evaluation of expressions**

Evaluating an expression means to calculate its value by performing the operations specified by the operators and substituting any variables with their given values. This process involves applying operator precedence rules and associativity to ensure the correct order of operations.

Key Concepts:

- Expressions: Combinations of values (constants or variables), operators, and sometimes function calls that can be evaluated to produce a result.
- Operators: Symbols that specify operations to be performed (e.g., +, -, *, /).
- Operands: The values on which operators act.
- Operator Precedence: The order in which operators are evaluated (e.g., multiplication before addition).
- Associativity: The direction (left-to-right or right-to-left) in which operators of the same precedence are evaluated.

Steps for Evaluating Expressions:

1. Substitution: Replace any variables in the expression with their given numerical values.
2. Operator Precedence: Identify operators with higher precedence and evaluate them first.
3. Associativity: For operators with the same precedence, apply the appropriate associativity rule (left-to-right or right-to-left).
4. Simplify: Continue evaluating sub-expressions and simplifying the expression until a single numerical value is obtained.

Example:

Consider the expression: 2 + 3 * 4

1. No variables to substitute.
2. Operator precedence: Multiplication has higher precedence than addition, so 3 * 4 is evaluated first, resulting in 12.
3. Expression becomes: 2 + 12
4. Final result: 14

In Programming:

- Expressions are fundamental to programming, allowing for calculations and decision-making.
- Programming languages have specific rules for operator precedence and associativity that govern how expressions are evaluated.
- Expression evaluation is crucial for assigning values to variables, controlling program flow, and performing calculations.

Type conversions in expressions

Type conversion, also known as type casting or coercion, is the process of changing a value's data type within an expression. This can happen automatically by the compiler (implicit conversion) or be specified by the programmer (explicit conversion). It's essential for performing operations involving different data types and ensuring correct calculations.

Implicit Type Conversion:

- Occurs automatically when an expression involves different data types.
- The compiler handles the conversion based on predefined rules, often promoting smaller data types to larger ones to avoid data loss according to Naukri.com.
- Example: In an expression like int + float, the int might be implicitly converted to a float before the addition is performed.

Explicit Type Conversion (Type Casting):

- The programmer explicitly specifies the desired data type using casting operators.
- This allows for more control over the conversion process, even if it might lead to data loss or unexpected results.
- Example: (float) int_variable casts the int_variable to a float.

Why Type Conversion is Necessary:

1. 1. Mixed-Type Operations:

When different data types are used in the same expression (e.g., adding an integer and a string), type conversion ensures that the operation can be performed.

2. 2. Data Type Compatibility:

Some functions or operations require specific data types. Type conversion allows you to adapt a value to the expected type.

3. 3. Avoiding Errors:

In some cases, type conversion can prevent runtime errors or unexpected behavior by ensuring that data types are compatible.

4. 4. Memory Management:

Converting a larger data type to a smaller one might be necessary to conserve memory or fit within a specific data structure.

Example in C:

```
#include <stdio.h>

int main() {
   int num_int = 10;
   float num_float = 3.14;

   // Implicit conversion: int to float in the addition
   float sum_implicit = num_int + num_float;
   printf("Implicit conversion: %f\n", sum_implicit);

   // Explicit conversion: float to int (truncation)
```

```
    int num_int_from_float = (int)num_float;
    printf("Explicit conversion: %d\n", num_int_from_float);

    return 0;
}
```

Unit 2 (8 Hours)

1. Control structures

The **conditional statements** (also known as decision control structures) such as if, if else, switch, etc. are used for decision-making purposes in C/C++ programs.
They are also known as Decision-Making Statements and are used to evaluate one or more conditions and make the decision whether to execute a set of statements or not. These decision-making statements in programming languages decide the direction of the flow of program execution.

Types of Conditional Statements in C/C++

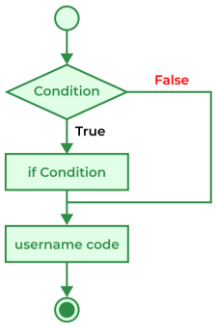
1. **if Statement**
2. **if-else Statement**
3. **Nested if Statement**
4. **if-else-if Ladder**
5. **switch Statement**
6. **Conditional Operator**
7. **Jump Statements:**
   - **break**
   - **continue**
   - **goto**
   - **return**

a. Decision statements; if and switch statement

The if statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statements is executed otherwise not.

Syntax of if Statement

```
if(condition)
{
  // Statements to execute if
  // condition is true
}
```



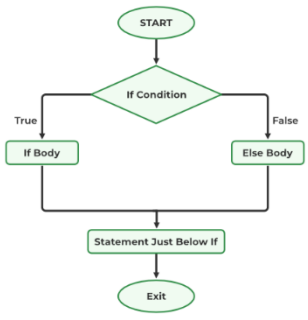
If- Else Statement

The *if* statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else when the condition is false? Here comes the C *else* statement. We can use the *else* statement with the *if* statement to execute a block of code when the condition is false. The if-else statement consists of two blocks, one for false expression and one for true expression.

Syntax of if else in C/C++

```
if (condition)
{
   // Executes this block if
   // condition is true
}
else
{
   // Executes this block if
  // condition is false
}
```

Flowchart of if-else Statement



Nested if statement

A nested if in C is an if statement that is the target of another if statement. Nested if statements mean an if statement inside another if statement. Yes, both C and C++ allow us to nested if statements within if statements, i.e, we can place an if statement inside another if statement.

Syntax of Nested if-else

```
if (condition1)
        {
    // Executes when condition1 is true
    if (condition2)

        // Executes when condition2 is true
    }
    else
    {
        // Executes when condition2 is false
    }
```

Else if Ladder

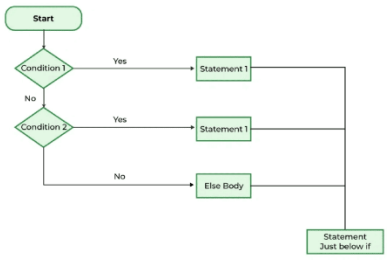
The if else if statements are used when the user has to decide among multiple options. The C if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the C else-if ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. if-else-if ladder is similar to the switch statement.

Syntax of if-else-if Ladder

```
if (condition)
    statement;
else if (condition)
    statement;
.
.

else
    statement;
```

Flowchart of if-else-if Ladder



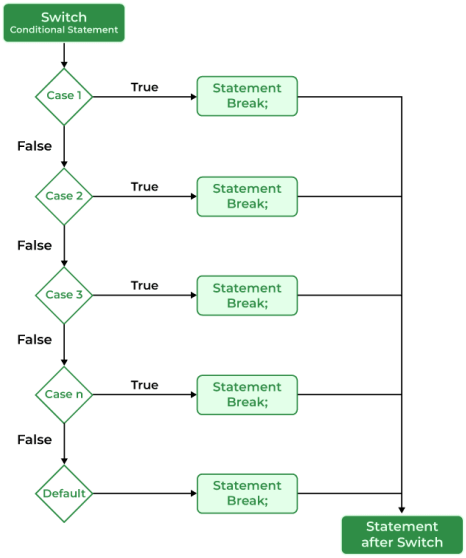
Switch statement

The switch case statement is an alternative to the if else if ladder that can be used to execute the conditional code based on the value of the variable specified in the switch statement. The switch block consists of cases to be executed based on the value of the switch variable.

Syntax of switch

```
        switch (expression)
case value1:
                statements;
   case value2:
                statements;
              ....
              ....
              ....
              default:
                statements;
          }
```

Flowchart of switch



➢ Loop control statements: while, for and do while loops

- Execution of a statement or set of statement repeatedly is called as looping.
- The loop may be executed a specified number of times and this depends on the satisfaction of a test condition.
- A program loop is made up of two parts one part is known as body of the loop and the other is known as control condition.
- Depending on the control condition statement the statements within the loop may be executed repeatedly

For

This is an entry controlled loop which provides compact loop control structure. In this, the initialization of loop control variable, the test condition and change in the value of loop control variable is done in the single statement separated by semicolon.

Syntax:

for (initialization; test condition; increment)
{
body of the loop;
}

While

The basic format of the while statement is as given below

while (test condition)
{
body of the loop;
}

This is an entry controlled loop. In this the test condition is placed before the body of the loop. The test condition is evaluated first and if it is true, then only the body of the loop will be executed.

- In the body of the loop there will be statement, which will alter the value of the loop control variable. After the execution of the body of the loop the test condition is again evaluated and if it is true then only the body of the loop will be executed. Then the body of the loop will be executed till the test condition becomes false.

- Once the test condition fails the statement following the body of the loop will be executed. The body of the loop will begin with a opening brace and ends with a closing brace. The body of the loop may have one or more statements.
  The loop control variable is used in the test condition to check whether the body of the loop is to be executed or not.

If the Test condition fails in the beginning itself the body of the loop will never be executed.

Do while loop

In some occasions it might be necessary to execute the body of the loop before the test is performed. Such situations can be handled with the help of do statement.

Syntax:

do
{
body of the loop;
}
while (test condition);

When the do statement is encountered, the body of the loop is executed. The test condition is present at the end of the body of the loop. The test condition is evaluated and if it is true, the body of the loop is executed again. This process

will continue till the test condition becomes false. Now the statement next to the test condition will be executed. Since the test condition is placed at the end of the body of the loop, this do loop is called as exit controlled loop. Therefore, even though the test condition fails at the first attempt itself, the body of the loop is executed at least once.

- **Example**:

```
i=0;
do
{
printf ("%d\n", i);
i++;
} while (i< 10);
```

jump statements, break, continue, goto statements

Transferring the execution control from one statement to another statement within the loop or from within the loop to outside the loop is possible in C. This kind of jumps can be achieved by break, goto and continue statements.

Break

Using this statement an early exit from a loop can be achieved. That is when the break statement is encountered within a loop then the loop is exited immediately and the statements following the loop will be executed. When a break is encountered within a nested loop, the loop in which this statement is present only that loop will be exited. That means the break will exit only one loop.

Syntax:

```
break;
(a) i=0;
while (i<10>
{
:
:
if (condition)
break;
:
:
}
n=7;
(b) i=0;
do
{
:
:
if (condition)
break;
:
:
```

```
} while (i<10);
n=7;
(c) while (i<15)
{
:
for (j=0; j<10; j++)
{
:
if (condition)
break;
:
}
n=7; (d) for (i=0; i<10; i++)
{
:
if(condition)
break;
}
n=7;
```

In all the above cases the break statement will make the control exit the loop in which it is residing. Thus it executes the statement n=7 in all the cases.
Since goto can be used to transfer the control to any place in a program it can be used to provide a branching within a loop or exit from deeply nested loop. A simple break statement could not work here.

Example 1:

```
while (i<15)
{
:
:
for (j=0; j<10; j++)
{
:
:
if (condition)
goto xyz;
:
:
xyz:
r=4
}
n=7;
:
```

}
m=8;

Continue statement

In a loop on some occasions depending on some condition we may have to skip a part of the body of the loop and proceed with the next iteration.
This can be achieved by the statement continue.

This loop control statement is just like the break statement. The *continue* statement is opposite to that of the break *statement*, instead of terminating the loop, it forces to execute the next iteration of the loop. As the name suggests the continue statement forces the loop to continue or execute the next iteration. When the continue statement is executed in the loop, the code inside the loop following the continue statement will be skipped and the next iteration of the loop will begin.

Syntax of continue

Syntax:

continue;

Example 1:

```
while (i<10)
{
:
if (condition)
continue;
:
}
n=7;
Example 2:
do
{
:
if (condition)
continue;
:
} while (i<10);
while (i<15)
{
:
for (j=0; j<10; j++)
{
:
if (condition)
continue;
:
}
:
}
```

Goto statement

goto

The goto statement in C/C++ also referred to as the unconditional jump statement can be used to jump from one point to another within a function.

Syntax of goto

```
Syntax1     |  Syntax2
----------------------------
goto label;  |   label:
.            |   .
.            |   .
.            |   .
label:       |   goto label;
```

In the above syntax, the first line tells the compiler to go to or jump to the statement marked as a label. Here, a label is a user-defined identifier that indicates the target statement. The statement immediately followed after 'label:' is the destination statement.

Jump Statement

These statements are used in C or C++ for the unconditional flow of control throughout the functions in a program. They support four types of jump statements:

➢ Arrays:

An array is defined as the collection of similar type of data items stored at contiguous memory locations. Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc. It also has the capability to store the collection of derived data types, such as pointers, structure, etc. The array is the simplest data structure where each data element can be randomly accessed by using its index number.

C array is beneficial if you have to store similar elements. For example, if we want to store the marks of a student in 6 subjects, then we don't need to define different variables for the marks in the different subject. Instead of that, we can define an array which can store the marks in each subject at the contiguous memory locations.

By using the array, we can access the elements easily. Only a few lines of code are required to access the elements of the array.

➢ Properties of Array

The array contains the following properties.

Each element of an array is of same data type and carries the same size, i.e., int = 4 bytes.

- Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.
- Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

Advantage of C Array

1) Code Optimization: Less code to the access the data.

2) Ease of traversing: By using the for loop, we can retrieve the elements of an array easily.

3) Ease of sorting: To sort the elements of the array, we need a few lines of code only.

4) Random Access: We can access any element randomly using the array.

Disadvantage of C Array

1) Fixed Size: Whatever size, we define at the time of declaration of the array, we can't exceed the limit. So, it doesn't grow the size dynamically like LinkedList which we will learn later.

Declaration of C Array

We can declare an array in the c language in the following way.

```
1. data_type array_name[array_size];
```

Now, let us see the example to declare the array.

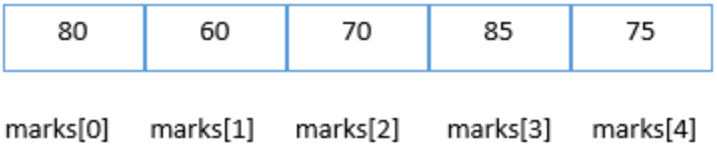
```
1. int marks[5];
```

Here, int is the *data_type*, marks are the *array_name*, and 5 is the *array_size*.

Initialization of C Array

The simplest way to initialize an array is by using the index of each element. We can initialize each element of the array by using the index. Consider the following example.

```
1. marks[0]=80;//initialization of array
2. marks[1]=60;
3. marks[2]=70;
4. marks[3]=85;
5. marks[4]=75;
```



Initialization of Array

C array example

```
#include<stdio.h>
int main(){
int i=0;
int marks[5];//declaration of array
marks[0]=80;//initialization of array
marks[1]=60;
marks[2]=70;
marks[3]=85;
marks[4]=75;
//traversal of array
for(i=0;i<5;i++){
    printf("%d \n",marks[i]);
    }//end of for loop
    return 0;
    }
```

➢ One dimensional array

A one-dimensional (1D) array is a linear data structure that stores a collection of elements of the same data type in contiguous memory locations. It's essentially a list where each element can be accessed by its index, starting from 0.

Key Characteristics:

- **Linear Structure:**

Elements are arranged sequentially, making it easy to traverse and access them one after another.

- **Homogeneous Data:**

All elements in the array must be of the same data type (e.g., all integers, all characters, etc.).

- **Contiguous Memory:**

The array's elements are stored in adjacent memory locations, allowing for efficient access using the index.

- **Index-Based Access:**

Each element is identified by its position (index) within the array, starting from 0.

Example (C):

```
int arr = {10, 20, 30, 40, 50};
// 'arr' is a 1D array of integers
// It has 5 elements, accessible by indices 0 through 4
```

➢ Declaration and initialization of one dimensional arrays,

In C, a one-dimensional array is declared and initialized using the syntax data_type array_name[array_size] = {value1, value2, ...};. The data_type specifies the type of elements (e.g., int, char, float), array_name is the identifier for the array, and array_size defines the number of elements the array can hold. The values within the curly braces {} are used to initialize the array elements during declaration.

Declaration and Initialization:

1. **Declaration:**

Declaring an array means reserving a block of memory to store a specific number of elements of the same data type. For example, int numbers[5]; declares an integer array named numbers that can hold 5 integer values.

2. **Initialization:**

Initialization provides initial values to the array elements. This can be done at the time of declaration (compile-time initialization) or later during program execution (runtime initialization).

Example:

```
#include <stdio.h>

int main() {
  // Compile-time initialization of an integer array
  int numbers[5] = {10, 20, 30, 40, 50};

  // Compile-time initialization of a character array (string)
  char message[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

  // Alternatively, a string literal can be used for character arrays
  char message2[6] = "Hello";
```

```
 // Runtime initialization using a loop
 int ages[3];
 printf("Enter three ages:\n");
 for (int i = 0; i < 3; i++) {
   scanf("%d", &ages[i]);
 }

 // Accessing and printing array elements
 printf("Numbers array: ");
 for (int i = 0; i < 5; i++) {
   printf("%d ", numbers[i]);
 }
 printf("\n");

 printf("Message: %s\n", message);
 printf("Message2: %s\n", message2);

 printf("Ages array: ");
 for (int i = 0; i < 3; i++) {
   printf("%d ", ages[i]);
 }
 printf("\n");

 return 0;
}
```

➢ two dimensional arrays,

A 2D array, also known as a two-dimensional array, is a data structure that organizes elements in a grid-like format with rows and columns. It's essentially an array of arrays, allowing for the representation of tabular data or matrices.

Key Characteristics:

- Structure: A 2D array is composed of rows and columns, similar to a table.
- Indexing: Elements are accessed using two indices: one for the row and one for the column.
- Applications: 2D arrays are widely used to represent:
    - Matrices: For mathematical operations and computations.
    - Tables: To store and organize data in a structured format.
    - Game boards: Representing the state of a game like chess or checkers.
    - Grids: For representing spatial data or layouts.

```
• C
• int matrix = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
  };
```

- This code declares a 2D array named matrix with 3 rows and 4 columns. The element at matrix would be 7

➢ Multi-dimensional arrays

Multi-dimensional arrays are data structures that store elements in a grid-like format with more than one dimension. They extend the concept of a one-dimensional array (a simple list) to allow for the organization of data in multiple dimensions, like rows and columns (2D) or even higher orders (3D, 4D, etc.).

Key characteristics:

- **Arrays of Arrays:**

Multi-dimensional arrays are fundamentally arrays that contain other arrays as their elements, creating nested structures.

- **Rows and Columns (2D):**

A two-dimensional array can be visualized as a table or matrix with rows and columns.

- **Higher Dimensions:**

Three-dimensional (3D) arrays can be thought of as a cube or a stack of matrices, and even higher dimensions are possible.

- **Indexing:**

Elements within a multi-dimensional array are accessed using multiple index values, one for each dimension. For example, in a 2D array, you might use array[row_index][column_index].

Common Use Cases:

- **Matrices and Linear Algebra:**

2D arrays are fundamental for representing and manipulating matrices in mathematical computations.

- **Image Processing:**

Images can be represented as 2D arrays where each element represents a pixel's color or intensity.

- **Game Development:**

Game worlds, character positions, and other game data can be stored in multi-dimensional arrays.

- **Data Analysis and Machine Learning:**

Multi-dimensional arrays are used extensively in handling and processing large datasets.

Examples:

- 2D Array (Matrix): [[1, 2, 3], [4, 5, 6], [7, 8, 9]] is a 2D array with 3 rows and 3 columns.
- 3D Array (Cube): Imagine a Rubik's Cube, where each face is a 2D array and the entire cube is a 3D array.

In essence, multi-dimensional arrays provide a powerful way to represent and work with structured data that has multiple relationships or characteristics, extending beyond the simple linear organization of a single-dimensional array.

3. Functions:

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

➢ Predefined Functions

So it turns out you already know what a function is. You have been using it the whole time while studying this tutorial! For example, main() is a function, which is used to execute code, and printf() is a function; used to output/print text to the screen:

Example

```
int main() {
  printf("Hello World!");
  return 0;
}
```

➢ Create a Function

To create (often referred to as *declare*) your own function, specify the name of the function, followed by parentheses `()` and curly brackets `{}`:

Syntax

```
void myFunction() {
  // code to be executed
}
```

➢ Example Explained

- myFunction() is the name of the function
- void means that the function does not have a return value. You will learn more about return values later in the next chapter
- Inside the function (the body), add code that defines what the function should do

➢ Call a Function

Declared functions are not executed immediately. They are "saved for later use", and will be executed when they are called. To call a function, write the function's name followed by two parentheses `()` and a semicolon `;` In the following example, `myFunction()` is used to print a text (the action), when it is called:

Example

Inside `main`, call `myFunction()`:

```
// Create a function
void myFunction() {
  printf("I just got executed!");
}

int main() {
  myFunction(); // call the function
  return 0;
}
```

- **Math Functions**

There is also a list of math functions available, that allows you to perform mathematical tasks on numbers. To use them, you must include the `math.h` **header file** in your program:

```
#include <math.h>
```

- Square Root

To find the square root of a number, use the `sqrt()` function:

Example

```
printf("%f", sqrt(16));
}
```

- **Round a Number**

The `ceil()` function rounds a number upwards to its nearest integer, and the `floor()` method rounds a number downwards to its nearest integer, and returns the result:

Example

```
printf("%f", ceil(1.4));
printf("%f", floor(1.4));
```

- Power

The `pow()` function returns the value of *x* to the power of *y* (x^y):

Example

```
printf("%f", pow(4, 3));
```

- **Other Math Functions**

A list of other popular math functions (from the `<math.h>` library) can be found in the table below:

Function	Description
abs(x)	Returns the absolute value of x
acos(x)	Returns the arccosine of x
asin(x)	Returns the arcsine of x
atan(x)	Returns the arctangent of x
cbrt(x)	Returns the cube root of x
cos(x)	Returns the cosine of x
exp(x)	Returns the value of E^x
sin(x)	Returns the sine of x (x is in radians)
tan(x)	Returns the tangent of an angle

C language has standard libraries that allow input and output in a program. The **stdio.h** or **standard input output library** in C that has methods for input and output.

- **scanf()**

The scanf() method, in C, reads the value from the console as per the type specified.

- **Syntax:**

scanf("%X", &variableOfXType);

where **%X** is the format specifier in C. It is a way to tell the compiler what type of data is in a variable and **&** is the address operator in C, which tells the compiler to change the real value of this variable, stored at this address in the memory.

- **printf()**

The printf() method, in C, prints the value passed as the parameter to it, on the console screen.

Syntax:

printf("%X", variableOfXType);

where **%X** is the format specifier in C. It is a way to tell the compiler what type of data is in a variable and **&** is the address operator in C, which tells the compiler to change the real value of this variable, stored at this address in the memory.

Unformatted and formatted I/O function in C

This article focuses on discussing the following topics in detail-

- Formatted I/O Functions.
- Unformatted I/O Functions.
- Formatted I/O Functions vs Unformatted I/O Functions.

Formatted I/O Functions

Formatted I/O functions are used to take various inputs from the user and display multiple outputs to the user. These types of I/O functions can help to display the output to the user in different formats using the format specifiers. These I/O supports all data types like int, float, char, and many more.
These functions are called formatted I/O functions because we can use format specifiers in these functions and hence, we can format these functions according to our needs.

List of some format specifiers-

S NO.	Format Specifier	Type	Description
1	%d	int/signed int	used for I/O signed integer value
2	%c	char	Used for I/O character value
3	%f	float	Used for I/O decimal floating-point value
4	%s	string	Used for I/O string/group of characters
5	%ld	long int	Used for I/O long signed integer value
6	%u	unsigned int	Used for I/O unsigned integer value

7	%i	unsigned int	used for the I/O integer value
8	%lf	double	Used for I/O fractional or floating data
9	%n	prints	prints nothing

1. printf()
2. scanf()
3. sprintf()
4. sscanf()

- **printf():**

printf() function is used in a C program to display any value like float, integer, character, string, etc on the console screen. It is a pre-defined function that is already declared in the stdio.h(header file).

Syntax 1:
To display any variable value.

printf("Format Specifier", var1, var2, ..., varn);

- scanf():

scanf() function is used in the C program for reading or taking any value from the keyboard by the user, these values can be of any data type like integer, float, character, string, and many more. This function is declared in stdio.h(header file), that's why it is also a pre-defined function. In scanf() function we use &(address-of operator) which is used to store the variable value on the memory location of that variable.

Syntax:
scanf("Format Specifier", &var1, &var2, ..., &varn);

- **sprintf():**

sprintf stands for "string print". This function is similar to printf() function but this function prints the string into a character array instead of printing it on the console screen.

Syntax:
sprintf(array_name, "format specifier", variable_name);

- **sscanf():**

sscanf stands for "string scanf". This function is similar to scanf() function but this function reads data from the string or character array instead of the console screen.

Syntax:
sscanf(array_name, "format specifier", &variable_name);

➢ Unformatted Input/Output functions

Unformatted I/O functions are used only for character data type or character array/string and cannot be used for any other datatype. These functions are used to read single input from the user at the console and it allows to display the value at the console.

Why they are called unformatted I/O?
These functions are called unformatted I/O functions because we cannot use format specifiers in these functions and hence, cannot format these functions according to our needs.

The following unformatted I/O functions will be discussed in this section-

1. getch()
2. getche()
3. getchar()
4. putchar()
5. gets()
6. puts()
7. putch()

Input Functions

➢ Getch()

getch() function reads a single character from the keyboard by the user but doesn't display that character on the console screen and immediately returned without pressing enter key. This function is declared in conio.h(header file). getch() is also used for hold the screen.

Syntax:
getch();
or
variable-name = getch();

➢ Getche()

getche():
getche() function reads a single character from the keyboard by the user and displays it on the console screen and immediately returns without pressing the enter key. This function is declared in conio.h(header file).

Syntax:

getche();
or
variable_name = getche();

➢ Getchar()

getchar():
The getchar() function is used to read only a first single character from the keyboard whether multiple characters is typed by the user and this function reads one character at one time until and unless the enter key is pressed. This function is declared in stdio.h(header file).

Syntax:

Variable-name = getchar();

- **Gets()**

Gets() function reads a group of characters or strings from the keyboard by the user and these characters get stored in a character array. This function allows us to write space-separated texts or strings. This function is declared in stdio.h(header file).

Syntax:

char str[length of string in number];
//Declare a char type variable of any length
gets(str);

- **Output functions**
- Putch()

Putch() function is used to display a single character which is given by the user and that character prints at the current cursor location. This function is declared in conio.h(header file)

Syntax:
putch(variable_name);

- putchar()

The putchar() function is used to display a single character at a time by passing that character directly to it or by passing a variable that has already stored a character. This function is declared in stdio.h(header file)

Syntax:
putchar(variable_name);

- Puts()

In C programming puts() function is used to display a group of characters or strings which is already stored in a character array. This function is declared in stdio.h(header file).

Syntax:
puts(identifier_name);

- **String Manipulation Function**

To solve this, C supports a large number of string handling functions in the standard library "string.h".

Few commonly used string handling functions are discussed below:

Function	Work of Function
strlen()	computes string's length
strcpy()	copies a string to another
strcat()	concatenates(joins) two strings
strcmp()	compares two strings
strlwr()	converts string to lowercase
strupr()	converts string to uppercase

➢ User defined and built-in Functions

A **user-defined function** is a type of function in C language that is defined by the user himself to perform some specific task. It provides code reusability and modularity to our program. User-defined functions are different from built-in functions as their working is specified by the user and no header file is required for their usage.
In this article, we will learn about **user-defined function, function prototype, function definition, function call, and different ways in which we can pass parameters to a function.**

How to use User-Defined Functions in C?

To use a user-defined function, we first have to understand the different parts of its syntax. The user-defined function in C can be divided into three parts:

1. Function Prototype
2. Function Definition
3. Function Call

➢ C Function Prototype

A function prototype is also known as a function declaration which specifies the **function's name, function parameters,** and **return type**. The function prototype does not contain the body of the function. It is basically used to inform the compiler about the existence of the user-defined function which can be used in the later part of the program.

Syntax

return_type **function_name** (type1 *arg1*, type2 *arg2*, ... typeN *argN*);

We can also skip the name of the arguments in the function prototype. So,

return_type **function_name** (type1 , type2 , ... typeN);

C Function Definition

Once the function has been called, the function definition contains the actual statements that will be executed. All the statements of the function definition are enclosed within **{ } braces.**

Syntax

return_type **function_name** (type1 *arg1*, type2 *arg2* .... typeN *argN*) {

```
   // actual statements to be executed
   // return value if any
}
```

Note: *If the function call is present after the function definition, we can skip the function prototype part and directly define the function.*

➢ C Function Call

C Function Call

In order to transfer control to a user-defined function, we need to call it. Functions are called using their names followed by round brackets. Their arguments are passed inside the brackets.

Syntax
function_name(arg1, arg2, ... argN);

➢ Components of Function Definition

There are three components of the function definition:

1. Function Parameters
2. Function Body
3. Return Value

➢ Passing Parameters to User-Defined Functions

We can pass parameters to a function in C using two methods:

1. Call by Value
2. Call by Reference

1. Call by value

In call by value, a copy of the value is passed to the function and changes that are made to the function are not reflected back to the values. Actual and formal arguments are created in different memory locations.

Note: *Values aren't changed in the call by value since they aren't passed by reference.*

2. Call by Reference

In a call by Reference, the address of the argument is passed to the function, and changes that are made to the function are reflected back to the values. We use the pointers of the required type to receive the address in the function.

- **Advantages of User-Defined Functions**

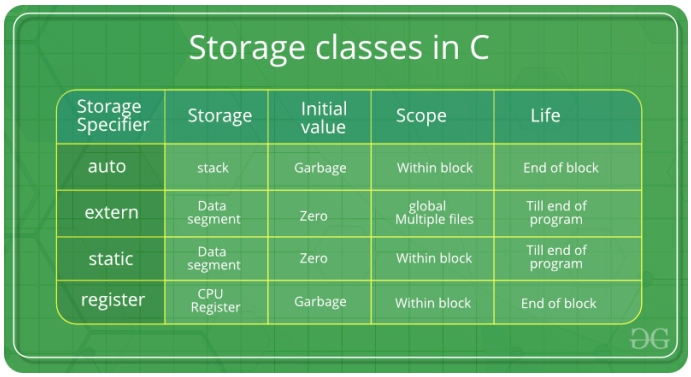
The advantages of using functions in the program are as follows:

- One can avoid duplication of code in the programs by using functions. Code can be written more quickly and be more readable as a result.
- Code can be divided and conquered using functions. This process is known as Divide and Conquer. It is difficult to write large amounts of code within the main function, as well as testing and debugging. Our one task can be divided into several smaller sub-tasks by using functions, thus reducing the overall complexity.
- For example, when using pow, sqrt, etc. in C without knowing how it is implemented, one can hide implementation details with functions.
- With little to no modifications, functions developed in one program can be used in another, reducing the development time.

➢ storage classes,

C Storage Classes are used to describe the features of a variable/function. These features basically include the scope, visibility, and lifetime which help us to trace the existence of a particular variable during the runtime of a program.

C language uses 4 storage classes, namely:



➢ Automatic

This is the default storage class for all the variables declared inside a function or a block. Hence, the keyword auto is rarely used while writing programs in C language. Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope). Of course, these can be accessed within nested blocks within the parent block/function in which the auto variable was declared. They are assigned a garbage value by default whenever they are declared.

➢ external (global)

Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different

block as well. So an extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere. It can be accessed within any function/block. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program.

➢ Static & registers

This storage class is used to declare static variables which are popularly used while writing programs in C language. Static variables have the property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope. So we can say that they are initialized only once and exist till the termination of the program. Thus, no new memory is allocated because they are not re-declared.

Their scope is local to the function to which they were defined. Global static variables can be accessed anywhere in the program. By default, they are assigned the value 0 by the compiler.

➢ Register

This storage class declares register variables that have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available. This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program.

If a free registration is not available, these are then stored in the memory only. Usually, a few variables which are to be accessed very frequently in a program are declared with the register keyword which improves the running time of the program. An important and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers.

➢ Register and static storage class

A storage class represents the visibility and a location of a variable. It tells from what part of code we can access a variable. A storage class in C is used to describe the following things:

- The variable scope.
- The location where the variable will be stored.
- The initialized value of a variable.
- A lifetime of a variable.
- Who can access a variable?

Thus a storage class is used to represent the information about a variable.

There are total four types of standard storage classes. The table below represents the storage classes in C.

Storage class	Purpose
auto	It is a default storage class.
extern	It is a global variable.

Storage class	Purpose
static	It is a local variable which is capable of returning a value even when control is transferred to the function call.
register	It is a variable which is stored inside a Register.

➢ Parameter passing in functions

The C function prototype is a statement that tells the compiler about the function's name, its return type, numbers and data types of its parameters. By using this information, the compiler cross-checks function parameters and their data type with function definition and function call.
Function prototype works like a function declaration where it is necessary where the function reference or call is present before the function definition but optional if the function definition is present before the function call in the program.

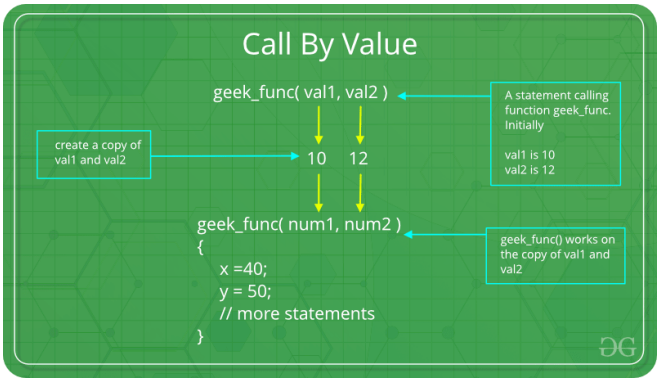
Syntax

return_type function_name(parameter_list);
where,

- return_type: It is the data type of the value that the function returns. It can be any data type int, float, void, etc. If the function does not return anything, void is used as the return type.
- function_name: It is the identifier of the function. Use appropriate names for the functions that specify the purpose of the function.
- parameter_list: It is the list of parameters that a function expects in parentheses. A parameter consists of its data type and name. If we don't want to pass any parameter, we can leave the parentheses empty.

Call by value

This method uses *in-mode* semantics. Changes made to formal parameters do not get transmitted back to the caller. Any modifications to the formal parameter variable inside the called function or method affect only the separate storage location and will not be reflected in the actual parameter in the calling environment. This method is also called ***call by value***.



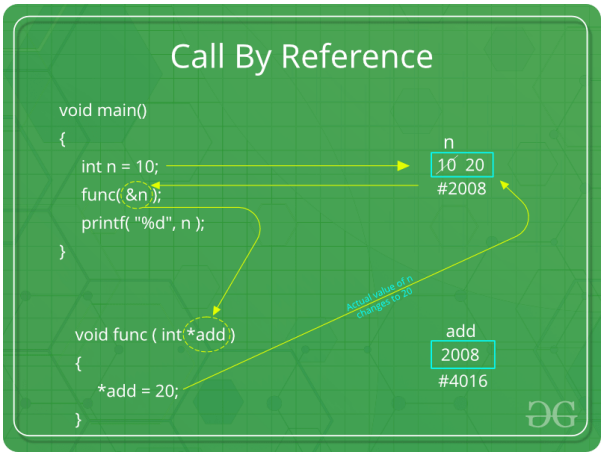
➢ **Shortcomings of Pass By Value:**

- Inefficiency in storage allocation
- For objects and arrays, the copy semantics are costly

4. Passing arrays to functions:

a. idea of call by reference

This technique uses *in/out-mode* semantics. Changes made to formal parameter do get transmitted back to the caller through parameter passing. Any changes to the formal parameter are reflected in the actual parameter in the calling environment as formal parameter receives a reference (or pointer) to the actual data. This method is also called as **call by reference**. This method is efficient in both time and space.



➢ **Shortcomings of Pass by Reference**

- Many potential scenarios can occur
- Programs are difficult to understand sometimes

5. Recursion

Recursion in C is a programming technique where a function calls itself during its execution. This allows complex problems to be broken down into smaller, similar subproblems, which are then solved by repeated calls to the same function.

Key Components of a Recursive Function:

- Base Case:

This is a crucial condition that defines when the recursion should stop. Without a base case, the function would call itself indefinitely, leading to an infinite loop and a stack overflow.

- Recursive Case:

This is the part of the function where it calls itself with a modified input, typically a smaller or simpler version of the original problem.

How Recursion Works:

When a recursive function is called, the current execution of the function is paused, and a new instance of the same function is created and executed. This process continues until the base case is reached. Once the base case is met, the function returns a value, and the execution unwinds back through the previous function calls, returning results at each step until the initial call completes.

Example: Factorial Calculation

```
#include <stdio.h>

long long factorial(int n) {
   // Base case: factorial of 0 or 1 is 1
   if (n == 0 || n == 1) {
      return 1;
   }
   // Recursive case: n * factorial of (n-1)
   else {
      return n * factorial(n - 1);
   }
}

int main() {
   int num = 5;
   printf("Factorial of %d is %lld\n", num, factorial(num)); // Output: Factorial of 5 is 120
   return 0;
}
```

- ### Advantages of Recursion:

- Elegance and Readability:

Recursive solutions can be more concise and easier to understand for problems that naturally exhibit a recursive structure (e.g., tree traversals, fractal generation).

- Problem Decomposition:

It simplifies complex problems by breaking them into smaller, manageable subproblems.

- ### Disadvantages of Recursion:

- Performance Overhead:

Recursive calls involve function call overhead (stack frame creation, parameter passing), which can be slower than iterative solutions for certain problems.

- Stack Overflow Risk:

Deep recursion can lead to a stack overflow error if the call stack exceeds its allocated memory limit.

- Debugging Complexity:

Debugging recursive functions can be more challenging due to the multiple function calls and the call stack.

➢ variable length character strings

In C, variable-length character strings are typically handled using character arrays and pointers, often in conjunction with dynamic memory allocation. While C does not have a built-in "string" data type that automatically resizes, it relies on conventions and functions to manage strings of varying lengths.

1. Null-Termination:

C strings are conventionally null-terminated, meaning a null character (\0) marks the end of the string. Functions like strlen(), strcpy(), and strcat() rely on this null terminator to determine string boundaries.

2. Character Arrays:

Strings are stored in arrays of char. When declaring a string using char array_name[size];, the size determines the maximum length the string can hold, including the null terminator.

3. Dynamic Memory Allocation:

For truly variable-length strings where the size is not known at compile time or needs to change during runtime, dynamic memory allocation functions like malloc(), calloc(), realloc(), and free() are used.

- malloc(): Allocates a block of memory of a specified size.
- realloc(): Resizes a previously allocated block of memory. This is crucial for handling strings that grow or shrink.
- free(): Deallocates memory previously allocated with malloc() or calloc(), preventing memory leaks.

Example of Dynamic String Handling:

```
#include <stdio.h>
#include <stdlib.h> // For malloc, realloc, free
#include <string.h> // For strlen, strcpy, strcat

int main() {
   char *myString = NULL; // Pointer to the string
   size_t currentSize = 0; // Current allocated size

   // Initial allocation
   myString = (char *)malloc(10 * sizeof(char));
   if (myString == NULL) {
      perror("Memory allocation failed");
      return 1;
   }
   strcpy(myString, "Hello");
   currentSize = 10;
   printf("Initial string: %s (allocated size: %zu)\n", myString, currentSize);

   // Reallocate to a larger size
   currentSize += 10;
```

```
    char *temp = (char *)realloc(myString, currentSize * sizeof(char));
    if (temp == NULL) {
        perror("Memory reallocation failed");
        free(myString); // Free original memory before exiting
        return 1;
    }
    myString = temp; // Update pointer to new memory location
    strcat(myString, ", World!");
    printf("After reallocation: %s (allocated size: %zu)\n", myString, currentSize);

    // Free the allocated memory
    free(myString);
    myString = NULL; // Set pointer to NULL after freeing

    return 0;
}
```

➢ inputting character strings

In C programming, inputting character strings refers to the process of reading a sequence of characters, typically from standard input (like the keyboard), and storing them into a character array, which represents a string in C. Unlike some other languages, C does not have a built-in "string" data type; instead, strings are handled as arrays of characters terminated by a null character (\0).

Several functions are available for inputting character strings in C:

- **scanf() with %s: This function reads a string until it encounters whitespace (space, tab, newline). It is suitable for single-word strings.**

```
#include <stdio.h>
int main() {
    char str[50];
    printf("Enter a word: ");
    scanf("%s", str); // Note: No '&' needed for arrays
    printf("You entered: %s\n", str);
    return 0;
}
```

- **gets(): This function reads an entire line of input, including spaces, until a newline character is encountered. However, gets() is considered unsafe due to potential buffer overflows and is generally discouraged.**

```
#include <stdio.h>
int main() {
    char str[50];
    printf("Enter a sentence (using gets): ");
    gets(str);
    printf("You entered: %s\n", str);
```

```
    return 0;
}
```

- **fgets(): This function is a safer alternative to gets(). It allows specifying the maximum number of characters to read, preventing buffer overflows. It also reads the newline character if present.**

```
#include <stdio.h>
int main() {
    char str[50];
    printf("Enter a sentence (using fgets): ");
    fgets(str, sizeof(str), stdin); // Reads up to sizeof(str)-1 characters or until newline
    printf("You entered: %s\n", str);
    return 0;
}
```

- **scanf() with a scanset (%[^...]): This allows reading characters until a specific character or set of characters is encountered. For instance, %[^\n] reads all characters until a newline, effectively reading a line with spaces.**

```
#include <stdio.h>
int main() {
    char str[50];
    printf("Enter a sentence (using scanf with scanset): ");
    scanf("%[^\n]", str); // Reads until newline
    // Optionally, consume the remaining newline character: scanf("%*c");
    printf("You entered: %s\n", str);
    return 0;
}
```

➢ Character library functions

Character library functions are a set of pre-defined functions in programming languages that are used to manipulate and work with individual characters or strings. These functions are typically found in header files like <ctype.h> in C/C++ and provide tools for tasks like character classification, conversion, and comparison.

1. Character Classification:

These functions determine the type of a character.

- Examples include:
    - isalpha(): Checks if a character is an alphabet (a-z, A-Z).
    - isdigit(): Checks if a character is a digit (0-9).
    - isspace(): Checks if a character is a whitespace character (space, tab, newline, etc.).
    - isupper(): Checks if a character is uppercase.
    - islower(): Checks if a character is lowercase.

2. Character Conversion:

These functions convert characters from one form to another.

- Examples include:
    - toupper(): Converts a lowercase character to uppercase.
    - tolower(): Converts an uppercase character to lowercase.

3. String Manipulation:

These functions operate on strings (sequences of characters).

- Examples include:
    - strlen(): Returns the length of a string (number of characters).
    - strcmp(): Compares two strings.
    - strcpy(): Copies one string to another.
    - strcat(): Concatenates (joins) two strings.
    - substr(): Extracts a substring from a string.

4. Other Useful Functions:

- getchar(): Reads a single character from the input (usually the keyboard).
- putchar(): Prints a single character to the output (usually the screen).
- sprintf(): Formats output into a string.
- sscanf(): Reads formatted input from a string.

These library functions are essential for various programming tasks, including data validation, text processing, and user input handling. By using these pre-built tools, programmers can save time and effort and focus on the core logic of their applications.

string handling functions

C programming language provides a set of standard library functions for manipulating strings, which are essentially arrays of characters terminated by a null character (\0). These functions are declared in the <string.h> header file.

- **strlen(const char *str):**

This function calculates and returns the length of the string str, excluding the null terminator. The return type is size_t.

- **strcpy(char *dest, const char *src):**

This function copies the string src (source) to the string dest (destination). It is crucial to ensure that dest has enough allocated memory to hold the copied string, including the null terminator, to prevent buffer overflows.

- **strncpy(char *dest, const char *src, size_t n):**

This function copies at most n characters from the string src to dest. If the length of src is less than n, null characters (\0) are appended to dest until n characters have been written. This function offers a safer alternative to strcpy when dealing with fixed-size buffers, as it prevents writing beyond the allocated memory.

- **strcat(char *dest, const char *src):**

This function concatenates (appends) the string src to the end of the string dest. The null terminator of dest is overwritten by the first character of src. Similar to strcpy, ensure dest has sufficient space.

- **strncat(char *dest, const char *src, size_t n):**

This function appends at most n characters from src to dest. It appends a null terminator after the appended characters.

- **strcmp(const char *str1, const char *str2):**

This function compares two strings, str1 and str2, lexicographically. It returns 0 if the strings are identical, a negative value if str1 is lexicographically less than str2, and a positive value if str1 is lexicographically greater than str2.

- **strncmp(const char *str1, const char *str2, size_t n):**

This function compares at most n characters of str1 and str2. The return values are similar to strcmp.

- **strstr(const char *haystack, const char *needle):**

This function searches for the first occurrence of the substring needle within the string haystack. It returns a pointer to the first character of the found substring, or NULL if needle is not found.